Journal of Universal Computer Science, vol. 13, no. 6 (2007), 874-890
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © *J.UCS*

Open and Closed Worlds for Overloading: a Definition and Support for Coexistence

Carlos Camarão
Universidade Federal de Minas Gerais
camarao@dcc.ufmg.br

Cristiano Vasconcellos
Universidade Federal de Pelotas
cristiano.damiani@ufpel.edu.br

Lucília Figueiredo
Universidade Federal de Ouro Preto
lucilia@dcc.ufmg.br

João Nicola
Universidade Federal de Minas Gerais
joaoraf@dcc.ufmg.br

Abstract The type system of Haskell and some related systems are based on an open world approach for overloading. In an open world, the principal type of each overloaded symbol must be explicitly annotated (in Haskell, annotations occur in type class declarations) and a definition of an overloaded symbol is required to exist only when overloading is resolved. In a closed world, on the other hand, each principal type is determined according to the types of definitions that exist in the relevant context and, furthermore, overloading resolution for an expression considers only the context of the definition of its constituent symbols. In this paper we formally characterize open and closed worlds, and discuss their relative advantages. We present a type system that supports both approaches together, and compare the defined system with Haskell type classes extended with multi-parameter type classes and functional dependencies. We show in particular that functional dependencies are not necessary in order to support multi-parameter type classes, and present an alternative route.

Key Words: Type system, type inference, constrained polymorphism, closed and open world approaches for overloading
Category: D.3.3, D.3.1

1 Introduction

The type system of Haskell [23, 12, 10, 20] and related type systems[31, 2, 26, 13, 4, 8] are based on an *open world* approach for overloading. In an open world, the principal type of each overloaded symbol must be explicitly annotated, and a definition of an overloaded symbol is required to exist only when overloading is resolved.

In Haskell, type annotations occur in *type class* declarations, and definitions of overloaded symbols are given in *instance* declarations.

For example, the principal types of `(==)` ("equal") and `(/=)` ("not equal") are annotated in type class *Eq* (defined in the Haskell prelude) as follows:

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

A class may contain, apart from type annotations of overloaded symbols, also *default definitions*, as shown in class *Eq* above. The following is an instance of class *Eq* for values of type *Int*, assuming that *primEqInt* is a function for comparing values of type *Int* for equality. The definition of `(/=)` in this instance is the default definition given in class *Eq*, since it is not explicitly given in the instance declaration.

```
instance Eq Int where
  (==) = primEqInt
```

In an open world, a definition of an overloaded symbol is required to exist only when overloading is resolved. For instance, no definitions of equality are required to be in the context where a definition of (polymorphic) equality of lists is given.

In a closed world, on the other hand, for any given expression the types of definitions available in the context where this expression occurs determine if the occurrence of this expression is well-typed or not and, in the first case, its principal type. A closed world is "closed" only in the level of modules, which introduce separate typing contexts. If, say, x is imported from a module M into another module M', then the uses of x in M' consider only the definitions of x that occur in M. If new definitions of x need to be given or used in M', an open world *must* be used. On the other hand, inside a module, a closed world is, in fact, "more open" than an open world, in the sense that a new definition of an overloaded symbol is not required to be an instance of any given annotated type. Each new definition of an overloaded symbol x implies a redefinition of x's principal type, as the least common generalization of the types of definitions of x in the typing context.

In this paper we construct a framework that allows us to give precise definitions of open and closed worlds, and discuss their relative advantages. A useful

result of this is the presentation of an alternative to the use of functional dependencies in an extension of Haskell with multi-parameter type classes. The paper starts by giving some preliminary definitions, in Section 2. Section 3 presents constraint-set satisfiability and simplification. In Section 4 we give formal definitions of open and closed worlds; relative advantages are compiled in subsection 4.1. Formal definitions of type systems to support both closed and open worlds are presented in Section 5. Inference of principal typings is discussed in Section 6, together with some relevant implementation issues. A brief discussion of ambiguity is given in Section 7. Section 8 concludes.

2 Preliminaries

We use types and terms of a language that is basically core-ML [21, 3, 22] extended with the possibility of introducing overloaded definitions in the outermost program scope, by means of a `leto` construct which does not introduce nested scopes. In this way, typing contexts are allowed to be stepwisely extended and may have more than one assumption for the same variable. The context-free syntax of expressions, their types and kinds of types is presented in Figure 2.

Programs	$p \ ::= e \mid \texttt{leto}\ x = e\ \texttt{in}\ p$	
Expressions	$e \ ::= x \mid \lambda x.\, e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	
Kinds	$\iota \ ::= \star \mid \iota_1 \to \iota_2$	
Simple type expressions	$\tau^\iota ::= \alpha^\iota \mid C^\iota \mid \tau_1^{\iota \to \iota'}\, \tau_2^\iota$	
Constraints	$\kappa \ ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	$(n \geq 0)$
Types	$\sigma \ ::= \tau \mid \kappa.\, \tau \mid \forall \alpha.\, \sigma$	

Figure 1: Context-free syntax of expressions and types

Each simple type expression has a *kind*, identified by an upper index in a simple type expression. A simple type is a type expression of kind $\star$, which is usually omitted. Meta-variables α, β, a, b, c dente type variables and C type constructors, of any kind.

Types of expressions are constrained polymorphic types. A set of constraints κ is a possibly empty set of pairs $x : \tau$, where x denotes an overloaded symbol and τ a simple type. A constrained polymorphic type is written as $\forall \alpha_1. \ldots . \forall \alpha_n.\, \kappa.\, \tau$, where $n \geq 0$. If $\kappa = \emptyset$, we have an unconstrained polymorphic type.

The set of free type variables of type σ, defined as usual, is denoted by $tv(\sigma)$.

We use $\forall \bar{\alpha}.\, \kappa.\, \tau$ as an abbreviation for $\forall \alpha_1. \cdots \forall \alpha_n.\, \kappa.\, \tau$, for some $n \geq 0$, and similarly for $\bar{\tau}$, $\bar{\sigma}$. Naturally, $\forall \bar{\alpha}.\, \sigma = \sigma$ if $n = 0$, and $\forall \bar{\alpha}.\emptyset.\tau = \forall \bar{\alpha}.\tau$. We also use

$\bar{\alpha}$ as a set of type variables, as in $\bar{\alpha} = tv(\tau)$.

A substitution S is a kind-preserving function from type variables to simple type expressions. The identity substitution is denoted by id. $S\sigma$ represents the capture-free[1] operation of substituting $S\alpha$ for each free occurrence of type variable α in σ. This operation is extended to constraints in the usual manner. We define $dom(S) = \{\alpha \mid S\alpha \neq \alpha\}$. It is sometimes convenient to use a finite mapping notation for substitutions, where $S = \{(\alpha_j \mapsto \tau_j)\}^{j=1..m}$ is used to denote the substitution such that $dom(S) = \{\alpha_j\}^{j=1..m}$ and $S\alpha_j = \tau_j$, for $j = 1, \ldots, m$. We also write $S \dagger \{\alpha_i \mapsto \tau_i\}^{i=1..n}$ to denote the substitution S' such that $S'\beta = S\beta$, if $\beta \notin \{\alpha\}^{i=1..n}$, and $S'\alpha_i = \tau_i$, for $i = 1, \ldots, n$. We define $\sigma[\bar{\tau}/\bar{\alpha}] = (id \dagger (\bar{\alpha} \mapsto \bar{\tau}))\sigma$.

In type systems with support for (universal) polymorphism, the type ordering is such that $\forall\alpha.\,\sigma \geq \sigma[\tau/\alpha]$, for *all* τ. The principal type of an expression in a typing context is the least upperbound, in this ordering, of all types that can be derived for this expression in this typing context. If $\sigma \geq \sigma'$, then σ' is called a *generic instance* of σ.

3 Constrained Polymorphism, Satisfiability and Simplification

In type systems with support for overloading, a typing context may include multiple assumptions for an overloaded symbol. The set of valid type assumptions which constitute a typing context is usually restricted by an overloading policy.

The principal type of an overloaded symbol x is obtained from the least common generalization (*lcg*) of the set of types in assumptions for x in the relevant typing context (unless the principal type of x is explicitly annotated). We say "the" least, instead of "a" least, by considering polymorphic types as equivalent up to renaming of bound type variables. Let $\tau \geq_S \tau'$ if $S\tau' = \tau$, and also $\tau \geq \tau'$ if there exists S such that $\tau \geq_S \tau'$. The *lcg* of a set of types $\{\sigma_i = \forall\bar{\alpha}_i.\,\kappa_i.\,\tau_i\}^{i=1..n}$ is $\forall\bar{\alpha}.\,\tau$, where $\bar{\alpha} = tv(\tau)$ and: i) $\tau_i \geq \tau$ (i.e. $\forall\bar{\alpha}.\,\tau \geq \forall\alpha_i.\,\tau_i$, $\bar{\alpha}_i = tv(\tau_i)$, for $i = 1, \ldots, n$; and ii) $\tau_i \geq \tau'$ implies $\tau \geq \tau'$ (i.e. $\forall\bar{\beta}.\tau' \geq \forall\bar{\alpha}.\,\tau$, where $\bar{\beta} = tv(\tau')$).

The *constrained least common generalization* of the types of x in a typing context Γ is the type $\sigma = \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$, where $\forall\bar{\alpha}.\,\tau$ is the lcg of $\Gamma(x)$ — written as $clcg(x, \Gamma, \sigma)$.

Example 1 Consider that the assumptions for `(==)` in a typing context $\Gamma_{(==)}$ are `(==)` : $Int \to Int \to Bool$ and `(==)` : $Float \to Float \to Bool$.

[1] The operation of applying substitution S to σ is capture-free if $tv(S\sigma) = tv(S(tv(\sigma)))$, where application of substitution S to a set of type variables $\{\alpha_i\}^{i=1..n}$ is given by $\{S\alpha_i\}^{i=1..n}$.

The following types can be derived for `(==)` in this typing context:

$$Int \to Int \to Bool,$$
$$Float \to Float \to Bool,$$
$$\forall a.\, \{\texttt{(==)} : a \to a \to Bool\}.\, a \to a \to Bool$$

The last one is the principal type of `(==)` in $\Gamma_{(==)}$. It can be instantiated to types of the form $\{\texttt{(==)} : \tau \to \tau \to Bool\}.\, \tau \to \tau \to Bool$, for which the constraint is *satisfiable* in Γ — in this particular case, τ can be either *Int* or *Float* or α, for some type variable α. If τ is *Int* or *Float*, the set of constraints can be simplified to an empty of constraints. □

Example 2 Consider the following definition of function *ins*, that uses `(==)`:

$$ins\ a\ [] \quad = [a]$$
$$ins\ a\ (b{:}x) = \texttt{if}\ a{==}b\ \texttt{then}\ b{:}x\ \texttt{else}\ b{:}ins\ a\ x$$

The principal type of this definition, obtained as the *clcg* of types of `(==)` in $\Gamma_{(==)}$, is: $\forall a.\, \{\texttt{(==)} : a \to a \to Bool\}.\, a \to [a] \to [a]$. Thus *ins* can be used in any context with a type that is an instance of $\forall \bar{\alpha}.\, \tau \to \tau \to Bool$, where $\bar{\alpha} = tv(\tau)$, if constraint-set $\{\texttt{(==)} : \tau \to \tau \to Bool\}$ is satisfiable in this context.

Example 3 (Functions overloaded over type constructors) Assume that there exist distinct definitions of function *ins*, in a typing context Γ_{ins}, for inserting elements in lists and trees, with types $\forall a.\, \{\texttt{(==)} : a \to a \to Bool\}.\, a \to [a] \to [a]$, and $\forall a.\, \{\texttt{(==)} : a \to a \to Bool\}.\, a \to Tree\ a \to Tree\ a$. In this context, the principal type of *ins* is $\forall a.\, \forall c.\, \{ins : a \to c\ a \to c\ a\}.\, a \to c\ a \to c\ a$, where c is a type variable of kind $\star \to \star$. Note that this type does not contain a constraint on `(==)`. Such constraint is automatically recovered from constraints on types of assumptions for *ins* in Γ_{ins}, if and when overloading is resolved, thus creating automatically a hierarchy of dependencies between overloaded symbols. □

Example 4 Consider that we also want to overload *ins* in typing context Γ'_{ins} by including definitions with the following types, in addition to those of Γ_{ins} of Example 3, that include a comparison operator as argument for working on ordered lists and trees (this could not be in Haskell extended with MPTCs if the principal type of *ins* was fixed a priori in a type class as $\forall a.\, \forall c.\, a \to c\ a \to c\ a$):

$$ins : \forall a.\, (a \to a \to Bool) \to a \to [a] \to [a]$$
$$ins : \forall a.\, (a \to a \to Bool) \to a \to Tree\ a \to Tree\ a$$

In Γ'_{ins}, the type of *ins* is $\forall a, b, c.\, \{ins :\ a \to b \to c\}.\, a \to b \to c$, where a, b, c are respectively the *lcg*s of:

$$\begin{array}{llll} \{\ a, & a, & a \to a \to Bool, & a \to a \to Bool \quad \} \\ \{\ [a], & Tree\ a, & a, & a \quad \} \\ \{\ [a], & Tree\ a, & [a] \to [a], & Tree\ a \to Tree\ a\ \} \end{array}$$

3.1 Constraint-set satisfiability

A constraint-set κ on a type $\sigma = \forall \bar{\alpha}.\, \kappa.\, \tau$ restricts the set of types to which σ can be instantiated, on a given typing context Γ, according to the type assumptions in Γ for the overloaded symbols that occur in κ.

A definition of constraint-set satisfiability independent on provability in a type system was given in [1]. We present below a simpler definition ($\circ$ denotes composition):

Definition 1 (Constraint and constraint-set satisfiability)

$$\frac{}{\Gamma \models_{id} \emptyset} \qquad (\mathtt{SAT_0})$$

$$\frac{S\tau = S\tau' \quad \Gamma \models_{S'} S\kappa}{\Gamma \cup \{x : \forall \bar{\alpha}.\, \kappa.\, \tau'\} \models_{S' \circ S} \{x : \tau\}} \qquad (\mathtt{SAT_1})$$

$$\frac{\Gamma \models_S \{x : \tau\} \quad \Gamma \models_S \kappa}{\Gamma \models_S \kappa \cup \{x : \tau\}} \qquad (\mathtt{SAT_n})$$

A *constraint-set satisfiability problem* is a problem of determining, for a given pair (Γ, κ), where Γ is a typing context and κ is a constraint-set, whether $\Gamma \models_S \kappa$ is provable, for some substitution S, which is called a *solution* to the satisfiability problem.

A solution S is *principal* if for any other solution S' there exists a substitution R such that $S' = R \circ S$. The application of the principal solution followed by constraint-set simplification is called *improvement*[12, 14].

If $\Gamma \models_S \kappa$ is provable, for some S, then we write $\Gamma \models \kappa$, otherwise $\Gamma \not\models \kappa$.

The constraint-set satisfiability problem has been proved to be undecidable [29]. However, practical implementations have been used, that either restrict the set of types of overloading symbols that may be introduced in typing contexts, in order to guarantee decidability, or use an iteration limit in order to prevent nontermination [17, 19, 11, 1].

Let a solution S to a satisfiabilty problem of κ in Γ be *minimal* if, for any other solution S', $dom(S) \cap tv(\kappa) \subseteq dom(S') \cap tv(\kappa)$ (informally, a solution is minimal if it "modifies κ less" than any other solution).

Definition 2 (Overloading resolution) Let Γ be any typing context and κ be any constraint-set involving a constraint on x — i.e. let $\kappa = \{x : \tau\} \cup \kappa'$, for some constraint-set κ'. *Overloading of x is resolved*, in an expression that has a principal type with constraint-set κ, if there exists a minimal solution S to the satisfiability of κ in Γ such that $S\tau = S'\tau$, for all other minimal solutions S'. □

3.2 Constraint-set Simplification

Constraints can be simplified either by removal of resolved constraints or substitution of constraints. For instance, $\{(\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\}$ can be simplified to an empty set of constraints, in typing context $\Gamma_{(\texttt{==})}$ of Example 1, and $\{\mathit{ins} : \alpha \to [\alpha] \to [\alpha]\}$ can be simplified to $\{(\texttt{==}) : \alpha \to \alpha \to \mathit{Bool}\ \}$, in typing context Γ_{ins} of Example 3.

Simplification yields equivalent constraint-sets, where equivalence, in any given typing context Γ, is the reflexive, symmetric and transitive closure of the simplification relation $\Gamma \models \kappa \gg \kappa'$ defined below.

Definition 3 (Constraint-set Simplification)

$$\frac{\Gamma \cup \{x : \forall \bar{\alpha}.\, \kappa'.\, \tau'\} \models_S \{x : \tau\} \cup \kappa \qquad \Gamma \not\models \{x : \tau\} \cup \kappa}{\Gamma \cup \{x : \forall \bar{\alpha}.\, \kappa'.\, \tau'\} \models \{x : \tau\} \cup \kappa \gg S(\kappa \cup \kappa')}$$

The premise of the rule above implies that overloading of x is resolved.

3.3 Open world satisfiability

Open world constraints — i.e. constraints introduced due to the specification, by programmers, of the principal type of overloaded symbols — are tested for satisfiability only when overloading of some symbol must be resolved. This can be formalized by means of the constraint projection operation $\kappa|^*_{tv(\tau) \cup tv(\Gamma)}$, which returns all constraints with type variables "dependent" on type variables of τ and Γ, transitively.

Definition 4 (Constraint projection)

$$\kappa|_V = \{x : \tau \in \kappa \mid tv(\tau) \cap V \neq \emptyset\} \qquad \kappa|^*_V = \begin{cases} \kappa|_V & \text{if } tv(\kappa|_V) \subseteq V \\ \kappa|^*_{tv(\kappa|_V)} & \text{otherwise} \end{cases}$$

Definition 5 (Open world satisfiability) $\Gamma \models^V_S \kappa$ holds if $\Gamma \models_S \kappa' \gg \emptyset$, where $\kappa' = \kappa - \kappa|^*_{V \cup tv(\Gamma)}$. $\Gamma \models^V \kappa$ holds if $\Gamma \models^V_S \kappa$ holds, for some S. □

Example 5 Consider the definition h `=` g *True*, in typing context $\Gamma_g = \{g : \mathit{Bool} \to \mathit{Char},\ g : \mathit{Char} \to \mathit{Bool},\ \mathit{True} : \mathit{Bool}\}$. In a closed world approach, h has type *Char*. In an open world, consider for example that the definitions of g with

these types are instances of the multi-parameter type class `class` $G\ a\ b$ `where` g`::` $a \rightarrow b$. In this case, h has principal type $G\ Bool\ b \Rightarrow b$, where b is a fresh type variable (written $\forall\beta.\{g : Bool \rightarrow \beta\}.\beta$ in `CT`). The reason for this is that it is possible for h to be exported and used in a program context where other definitions of g exist, and one of these could be used in the evaluation of $g\ True$ (for example, a definition with type $Bool \rightarrow Int$). According to Definition 5, we have: $\Gamma \models^{\{\beta\}} \{g : Bool \rightarrow \beta\}$, for any Γ, that is, no definition of g is required to exist in order to type $g\ True$.

An extension of Haskell with functional dependencies allows programmers to "close the world". In this example, type variable b can be explicitly defined to "depend on a" (or a can be specified to determine b), by annotation of a functional dependency: `class` $G\ a\ b$ `|` a `->` b `where` g`::` $a \rightarrow b$. With such a functional dependency, the world is closed, i.e. satisfiability is checked in typing context Γ_g, returning a substitution that, applied to type $G\ Bool\ b \Rightarrow b$, improves (or simplifies) it to $Char$. □

4 Open and Closed Worlds: a Formal Definition

In this section we give formal definitions of open and closed worlds, based on the definitions given before. Our characterization is somewhat different from the one based on open and closed refinement kinds of Duggan and Ophel[7]. Refinement kinds are defined so as to a priori allow constrained polymorphic types to be "incrementally extended" (in the case of open kinds) or not (for closed kinds). Closed refinement kinds completely characterize ("close up"), by themselves, the possibly infinite set of instance types. For this, refinement kinds use intersection types and fixed point operators. In contrast, our definitions are always respective to a given typing context.

Definition 6 (Open World) An *open world* is characterized by:

1. The principal type of each overloaded symbol is specified by a single type annotation and $\Gamma \models^{tv(\tau_x)} \kappa_x$ holds, where Γ is the typing context and $x : \forall\bar{\alpha}.\kappa_x.\tau_x$ is the type assumption corresponding to this annotation.
2. In such Γ, each definition of x must have a type $\sigma = \forall\bar{\beta}.\kappa.\tau$ such that $\bar{\beta} = tv(\kappa.\tau)$, $\tau = S\tau_x$ and $\kappa \supseteq S\kappa_x$, for some S, and $\Gamma \models^{tv(\tau)} \kappa$.
3. $\Gamma \models^{tv(\tau)} \kappa$ holds for all Γ, κ, τ such that $\Gamma \vdash \kappa.\tau$ is derivable.

In $\Gamma \models^{tv(\tau)} \kappa$, the set $tv(\tau)$ can be restricted, e.g. by functional dependencies in Haskell with MPTcs, so as to close the world (as illustrated in Example 5). In this case, $tv(\tau)$ should be subtracted by a set of variables in κ that are specified

as targets (i.e. that occur at the right-hand side) of some functional dependency and for which the corresponding source type variables have been instantiated (i.e. do not occur in κ).

Definition 7 (Closed World) A *closed world* is characterized by:

1. The principal type of x in Γ is the *clcg* of the types of x in Γ.
2. A type annotation for e is an instance of e's principal type.
3. $\Gamma \models \kappa$ holds for all Γ, κ, τ such that $\Gamma \vdash \kappa.\tau$ is derivable.
4. If definitions of x are given in a module M — in which type assumptions are given by Γ — and x is imported into a module N, then $\Gamma(x)$ gives the types in type assumptions for x used to type uses of x in N.

4.1 Open versus closed

This section compiles relative advantages of open and closed worlds. For space reasons, we only include major issues. We start with the advantages of an open world:

- Overloaded symbols may be used without requiring definitions of overloaded symbols to be visible (they must be visible only when overloading is resolved).
- The type inference algorithm can behave more efficiently in an open world, due to satisfiability being tested only when overloading must be resolved. This is rather significant in the case of frequently used overloaded symbols (e.g. `(==)`).

However, in a closed world:

- Types of overloaded symbols need not be annotated. The annotation of constraints of types of overloaded symbols require that programmers decide, in advance, which particular definitions an overloaded symbol might possibly have (in Haskell, programmers must decide, in particular, how to organize type class hierarchies). In some situations, this requirement can be rather inconvenient (as pointed out by e.g. Odersky, Wadler and Wehr [24, page 137]). Since constraints include information that is dependent on the implementation of a function (i.e. on the fact that an implementation uses some particular overloaded symbols), if the implementation of a function is changed, so that the new implementation uses a different set of overloaded symbols, program parts that use this function need to be modified, if they include type annotations, even if the types of argument and result of the

function remain the same. In a closed world, the hierarchy of dependencies between overloaded symbols need not be given by programmers, but are recovered automatically by the types of overloaded symbols (see e.g. Example 3).

– A closed world allows the inference of more informative types, (possibly causing overloading to be resolved), due to earlier constraint-set satisfiability checking. Practical examples for which an analysis of the types of overloaded symbols available in a typing context can be used to instantiate the types of expressions that use these overloaded symbols can be found in e.g. [14, 8].

– A closed world approach opens possibilities for optimizations in code generation which do not depend (unlike the case of an open world approach [6, 18]) on a global analysis of the entire program for efficient dynamic dispatching to an appropriate definition of an overloaded symbol.

5 Type system

Figure 2 presents a version of system `CT` to support a closed world. This is extended in Section 5.1 in order to support both open and closed worlds together. Typing formulas $\Gamma \vdash e : \sigma$ express that expression e has type σ in typing context Γ. following are used in Figure

For any typing context Γ, $(\Gamma\,;\, x : \sigma)$ denotes $\Gamma \cup \{x : \sigma\}$ if $\Gamma \cup \{x : \sigma\}$ is a valid typing context (according to the adopted overloading policy), and $(\Gamma,\ x : \sigma) = (\Gamma \ominus x)\,;\, x : \sigma$, where $\Gamma \ominus x = \Gamma - \{x : \sigma \mid x : \sigma \in \Gamma\}$. Predicate $gen(\kappa.\,\tau, \sigma)$ is defined to hold if $\sigma = \forall\bar{\alpha}.\,\kappa.\,\tau$, where $\bar{\alpha} = tv(\kappa.\,\tau)$. Instantiation of constrained polymorphic types is restricted by constraint-set satisfiability:

Definition 8 (Instance relation of constrained types)

$$\frac{\sigma \geq \forall\bar{\alpha}.\,\kappa.\,\tau \qquad \Gamma \models \kappa}{\sigma \geq_\Gamma \forall\bar{\alpha}.\,\kappa.\,\tau}$$

Note that if $\Gamma(x)$ is a singleton $\{x : \tau\}$ then $clcg(x, \Gamma, \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau)$ and $clcg(x, \Gamma, \tau)$ hold, as $\forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$ and τ are equivalent modulo constraint-set simplification in Γ.

5.1 Selectively opening a closed world

An extension of system `CT` to support also an open world is presented in Figure 5.1. It is based on the use of special type annotations, that specify the types of overloaded symbols, as in the following example: `assume (==)` :: $a \to a \to Bool$.

$$\frac{clcg(x,\Gamma,\sigma) \quad \sigma \geq_\Gamma \kappa.\,\tau}{\Gamma \vdash x : \kappa.\,\tau} \qquad (\texttt{VAR})$$

$$\frac{\Gamma, x : \tau' \vdash e : \kappa.\,\tau}{\Gamma \vdash \lambda x.\, e :\ \kappa.\,\tau' \to \tau} \qquad (\texttt{ABS})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash e_1\, e_2 : \kappa_1 \cup \kappa_2.\,\tau_1} \quad \Gamma \models \kappa_1 \cup \kappa_2 \qquad (\texttt{APPL})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad (\texttt{LET})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma; x : \sigma_1 \vdash p : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad (\texttt{LETO})$$

Figure 2: Type system CT supporting a closed world

A clause assume$x :: \tau$ introduces in the typing context an *open-world assumption* $x : \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$, where $\bar{\alpha} = tv(\tau)$. Note that the form of an assume clause contributes to restoring data abstraction, since it does not include constraints.

The support for open world in system CT is based on characterizing type assumptions and constraints as open or closed. Open world type assumptions and constraints are introduced in a typing context only by means of assume clauses, and the second by lambda, let or leto bindings (type derivation for lambda and let bound variables are done in the same manner as for leto bound variables). We define that ow and cw filters out open and closed world assumptions, respectively, from a typing context or a constraint-set ($\Gamma = ow(\Gamma) \cup cw(\Gamma)$ and similarly for κ). A typing context Γ must be such that $ow(\Gamma)$ satisfies the requirements in Definition 6. $\Gamma(x)$ is redefined to mean $ow(\Gamma)(x)$ if $ow(\Gamma)(x) \neq \emptyset$, otherwise $cw(\Gamma)(x)$. The satisfiability relation $\Gamma \models^{*V} \kappa$ holds if both $\Gamma \models^V ow(\kappa)$ and $\Gamma \models cw(\kappa)$ hold. The instantiation relation is modified in order to use the combined satisfiability relation: $\dfrac{\sigma \geq \forall\bar{\alpha}.\,\kappa.\,\tau \qquad \Gamma \models^{*tv(\tau)} \kappa}{\sigma \geq_\Gamma \forall\bar{\alpha}.\,\kappa.\,\tau}$.

To summarize the modifications to support also an open world: i) a modified constraint-set satisfiability relation, on the side condition of rule $\texttt{APPL}_\text{o}$, considers the possibility of occurrence of open and closed constraints together and, for open world constraints, defers satisfiability to when overloading is resolved, ii) a correspondingly modified instantiation relation is used in rule $\texttt{VAR}_\text{o}$, and iii) a side condition in rule $\texttt{LETO}_\text{o}$ restricts the type of definitions of overloaded symbols to be an instance of its principal type, if this type is explicitly specified.

$$\frac{clcg(x,\Gamma,\sigma) \quad \sigma \geq_\Gamma \kappa.\,\tau}{\Gamma \vdash x : \kappa.\,\tau} \qquad (\texttt{VAR})$$

$$\frac{\Gamma, \{x : \tau'\} \vdash e : \kappa.\,\tau}{\Gamma \vdash \lambda x.\, e :\ \kappa.\,\tau' \to \tau} \qquad (\texttt{ABS})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash e_1\ e_2 : \kappa_1 \cup \kappa_2.\,\tau_1} \quad \Gamma \models^{*tv(\tau_1)} \kappa_1 \cup \kappa_2 \qquad (\texttt{APPL}_\text{o})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma, \{x : \sigma_1\} \vdash e_2 : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \kappa_2.\,\tau_2} \quad gen(\kappa_1.\,\tau_1, \sigma_1) \qquad (\texttt{LET})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma; \{x : \sigma_1\} \vdash p : \kappa_2.\,\tau_2}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : \kappa_2.\,\tau_2} \quad \begin{array}{l} gen(\kappa_1.\,\tau_1, \sigma_1) \\ ow(\Gamma)(x) = \{\sigma\} \text{ implies } \sigma \geq_\Gamma \sigma_1 \end{array} \qquad (\texttt{LETO}_\text{o})$$

Figure 3: Type system CT supporting open and closed worlds

6 Type Inference

A prototype implementation of the front-end of a compiler for a language that is essentially "Haskell without type classes" but with support for constrained polymorphism as defined in system CT, is available at:

`http://www.dcc.ufmg.br/~camarao/CT/`

The type inference algorithm in this prototype supports both open and closed worlds and is an extension — to support (possibly mutually recursive) binding groups — of the algorithm defined in Figure 6 . This algorithm is defined as a syntax-directed proof system of judgements $\Gamma \vdash e : (\kappa.\,\tau, \Gamma')$, where $(\forall\bar{\alpha}.\,\kappa.\,\tau, \Gamma')$ is the principal typing of e in Γ [9, 28], where $\bar{\alpha} = tv(\kappa.\,\tau) - tv(\Gamma)$. The algorithm uses the following functions.

Function $clcg_a$, for computing the constrained least common generalization of the set of types of some symbol x in a typing context Γ, is given by $clcg_a(x,\Gamma) = \forall\bar{\alpha}.\,\{x : \tau\}.\,\tau$, where $\forall\bar{\alpha}.\,\tau = lcg_a(\Gamma(x))$. Function lcg_a computes the *lcg* of a given set of types, based on a function for computing the *lcg* of any set of simple type expressions. Finite mappings $\mathcal{S}$ from type variables to pairs of simple types are used in lcg' to "remember" generalizations already performed. For example, *lcg* applied to the set of types $\{\alpha_1 \to \beta_1 \to \alpha_1,\ \alpha_2 \to \beta_2 \to \alpha_2\}$ gives $\alpha \to \beta \to \alpha$, where α, β are fresh (and not, say, $\alpha \to \beta \to \alpha'$). lcg' needs to remember generalizations only inside a pair of type expressions. χ is used to

denote a type variable or constructor.

$$lcg_a(\{\sigma_i\}^{i=1..n}) = \forall\bar{\alpha}.\,\tau, \text{ where } \sigma_i = \forall\bar{\alpha}.\kappa_i.\,\tau_i, \text{ for } i = 1,\ldots,n,\\ (\tau,\mathcal{S}) = lcg'(\{\tau_i\}^{i=1..n},\emptyset), \text{ for some } S, \text{ and}\\ \bar{\alpha} = tv(\tau)$$

$$lcg'(\{\tau\},\mathcal{S}) = (\tau,\mathcal{S})$$

$$lcg'(\{\chi_1^{\iota_1},\chi_2^{\iota_2}\},\mathcal{S}) = \begin{cases} (\chi_1^{\iota_1},\mathcal{S}) & \text{if } \chi_1 = \chi_2 \text{ (which implies } \iota_1 = \iota_2) \\ (\alpha,\mathcal{S}) & \text{if } \mathcal{S}(\alpha) = (\chi_1^{\iota_1},\chi_2^{\iota_2}), \text{ for some } \alpha \\ (\alpha',\mathcal{S}\dagger\{\alpha' \mapsto (\chi_1^{\iota_1},\chi_2^{\iota_2})\} & \text{otherwise, where } \alpha' \text{ is fresh} \end{cases}$$

$$lcg'(\{\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4}\},\mathcal{S}) =$$

if $S(\alpha) = (\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4})$, for some α, **then** $(\alpha,\mathcal{S})$
else if $i_1 \neq i_3$ or $i_2 \neq i_4$ **then** $(\alpha',\mathcal{S}\dagger\{\alpha' \mapsto (\tau_1^{\iota_1}\ \tau_2^{\iota_2},\ \tau_3^{\iota_3}\ \tau_4^{\iota_4})\})$, α' fresh
else $(\tau_a^{\iota_1}\ \tau_b^{\iota_2},\mathcal{S}_2)$, where: $(\tau_a^{\iota_1},\mathcal{S}_1) = lcg'(\{\tau_1^{\iota_1},\ \tau_3^{\iota_3}\},\mathcal{S})$
$(\tau_b^{\iota_2},\mathcal{S}_2) = lcg'(\{\tau_2^{\iota_2},\ \tau_4^{\iota_4}\},\mathcal{S}_1)$

$$lcg'(\{\tau_1,\tau_2\}\cup\mathcal{T},\mathcal{S}) = lcg'(\{\tau\}\cup\mathcal{T},id)$$
where $(\tau,\mathcal{S}') = lcg'(\{\tau_1,\tau_2\},id)$

Function *simplify* simplifies constraint-sets, as defined in Section 3.2. It can be defined as follows.

$$simplify(\kappa,\Gamma) = simplify(\kappa,\Gamma,\emptyset)$$

$$simplify(\emptyset,\Gamma,\kappa_0) = \emptyset$$

$$simplify(\{o:\tau\},\Gamma,\kappa_0) = \textbf{if } tv(\tau) = \emptyset \textbf{ then } \emptyset \textbf{ else}\\ \textbf{if } o:\tau \in \kappa_0 \textbf{ then } \{o:\tau\} \textbf{ else}\\ \textbf{if } \exists \text{ a single } \forall\bar{\alpha}.\kappa'.\tau' \in \Gamma(o) \text{ s.t. } S\tau' = S\tau, \text{ for some } S\\ \textbf{then } simplify(S\kappa',\Gamma,\kappa_0\cup\{o:\tau\}) \textbf{ else } \{o:\tau\}$$

$$simplify(\{o:\tau\}\cup\kappa,\Gamma,\kappa_0) = simplify(\{o:\tau\},\Gamma,\kappa_0)\cup simplify(\kappa,\Gamma,\kappa_0)$$

Function *sat* implements a practical solution to the satisfiability problem by returning a principal solution whenever one exists and by using an iteration limit to stop, when it cannot find a solution. The definition of *sat* is given in [1].

Function *unify* implements unification[2] (see e.g. [22]). We assume that all types in outermost typing contexts are closed (otherwise unification would be needed in the case of leto-bindings, as done for let-bindings), and also use $st(x,\Gamma,\Gamma') = \{\tau = \tau' \mid x:\tau \in \Gamma, x:\tau' \in \Gamma'\}$ and $\Gamma^*_x = \{x:\sigma_i\}^{i=1..n} \cup \bigcup_{y:\tau\in\kappa_i,\text{ for some }\tau,i\in\{1..n\}} \Gamma^*_y$, where $\Gamma(x) = \{\sigma_i = \forall\bar{\alpha}_i.\,\kappa_i.\,\tau_i\}^{i=1..n}$.

Theorems on soundness and principal typing of the type inference algorithm (not even stated in this paper) can be proved by induction on the type inference

[2] Two types σ,σ' are unifiable if there exists a substitution S such that $S\sigma = S\sigma'$.

$$\frac{\forall\bar{\alpha}.\,\{x:\tau\}.\tau = clcg_a(x,\Gamma) \quad \kappa = simplify(\{x:\tau\},\Gamma)}{\Gamma \vdash x : (\kappa.\,\tau, \Gamma^*_x)}$$

$$\frac{\Gamma \vdash (e : \kappa.\,\tau, \Gamma')}{\Gamma \vdash \lambda x.\, e : (\kappa.\,\tau' \to \tau, \Gamma' \ominus x)} \qquad \tau' = \begin{cases} \tau_x \text{ if } x : \tau_x \in \Gamma' \\ \alpha \text{ otherwise, } \alpha \text{ fresh} \end{cases}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\,\tau_1, \Gamma_1) \quad \Gamma \vdash e_2 : (\kappa_2.\,\tau_2, \Gamma_2)}{\Gamma \vdash e_1\ e_2 : (S'S(\kappa_1 \cup \kappa_2.\,\alpha), S\Gamma_1 \cup S\Gamma_2))} \begin{array}{l} S = unify(\{\tau_1 = \tau_2 \to \alpha\} \cup st(x,\Gamma_1,\Gamma_2)) \\ S' = sat(\kappa_1 \cup \kappa_2, \Gamma),\ \alpha \text{ fresh} \end{array}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\,\tau_1, \Gamma_1) \quad \Gamma, \{x:\sigma_1\} \vdash e_2 : (\kappa_2.\,\tau_2, \Gamma_2)}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (S'S(\kappa_2.\,\tau_2), S\Gamma_2 \ominus x)} \begin{array}{l} S = unify(st(x,\Gamma_1,\Gamma_2)\}) \\ S' = sat(\kappa_1 \cup \kappa_2, \Gamma) \end{array}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\,\tau_1, \Gamma_1) \quad \Gamma; \{x:\sigma_1\} \vdash p : (\kappa_2.\,\tau_2, \Gamma_2)}{\Gamma \vdash \texttt{leto}\ x = e_1\ \texttt{in}\ p : (S(\kappa_2.\,\tau_2), \Gamma_2 \ominus x)} \begin{array}{l} S = sat(\kappa_1 \cup \kappa_2, \Gamma) \\ ow(\Gamma)(x) = \{\sigma\} \text{ implies } \sigma \geq_\Gamma \sigma_1 \end{array}$$

Figure 4: Algorithm for Inference of Principal Typings

rules, using proofs of correctness of *sat*, *simplify* and lcg_a. We are unfortunately still working on small details of the proofs.

The implementation of the type inference algorithm uses the core of an algorithm by Mark Jones [15] and is based also on other works by the authors, covering constraint-set satisfiability and polymorphic recursion [1, 28, 9]. We have very recently "glued" our front-end to GHC's[19] back-end, and a compiler will thus be publicly available soon.

Our experience with our prototype implementation, with respect to whether the iteration limit will only be exercised in rare cases in practice, is unfortunately rather limited up to now, because we have been mostly exercising the prototype with "desugared" versions of *valid* Haskell programs (with type classes replaced by assume clauses and instances becoming normal declarations).

Even if constrained types are restricted as e.g. in Haskell 98, where type classes can only have a single parameter, time complexity of satisfiability of a constraint on x still grows exponentially with the number of assumptions for x in the typing context [25, 30]. In cases of heavily overloaded symbols, the performance of the type inference algorithm thus degrades *if* an open world approach is not used (so as to restrict satisfiability to when overloading is resolved). A typical situation occurs with the overloading of `(==)`, which is defined for basic values (integers, floating point numbers etc.), lists, pairs, triples etc.

Recent work has been developed in order to define suitable conditions on types of overloaded symbols that guarantee decidability of type inference and are not too restrictive in practice [27, 5, 16]. However, we are reluctant at the moment to incorporate such restrictions, because of the added complexity to the

language, the abscence of a semantic characterization for such restrictions, and because restrictions prevent nontermination but not termination after a very long time (shouldn't the bottom line be lower/stricter?).

7 Ambiguity

A full discussion of type ambiguity and semantics coherence in the context of type systems with support for constrained polymorphism is, in our view, a subject that has not yet been investigated in sufficient depth. Usually, an expression e is considered *ambiguous* if two distinct denotations can be obtained for it, using a semantics defined inductively on the derivation of a type for e [22]. A so-called *coherent* semantics does not specify a meaning to such ambiguous expressions. With respect to derivations in the type system, we can translate this requirement so as to avoid the existence of two or more derivations of the same type for e that assign distinct non-unifiable types for some subterm of e. There is also a third, more syntactic characterization, which is the one we shall briefly consider below (again, we are not aware of any in-depth work relating any of these).

Consider a type with constraint-set κ and simple type τ, occurring in a typing context Γ. We can consider, in type systems for constrained polymorphism:

Definition 8 $\kappa.\tau$ is ambiguous if there exist two distinct minimal solutions S_1 and S_2 to the satisfiability of κ in Γ such that $S_1\tau = S_2\tau$. □

This definition can be relaxed, allowing a greater set of expressions to be well-typed, if we consider instead:

Definition 9 $\kappa.\tau$ is ambiguous if there exists a minimal solution S to the satisfiability of κ in Γ such that all other minimal solutions S' are such that $S\tau = S'\tau$. □

With Definition 9, an expression such as, for example, $f\ x$ is not considered ambiguous in a typing context containing $\{f : \mathit{Int} \to \mathit{Int}, f : \mathit{Int} \to \mathit{Float}, f : \mathit{Float} \to \mathit{Float}, x : \mathit{Int}, x : \mathit{Float}\}$. The motivation for Definition 9 is clear: even though there exist two distinct type derivations for $f\ x : \mathit{Float}$, it can be effectively used in a context requiring it to have type Int. It is worthwhile to note though that Definition 9 would contradict (in the example above) a usual definition of semantics coherence.

Though we haven't unfortunately proved it yet, we expect that ambiguity as given in Def. 9 is sufficient to reject all derivations that would lead to semantic incoherence.

In an open world, ambiguity must be tested, as satisfiability itself, only when the satisfiability condition indicates so. Also, the use of constraint-projection, on which the satisfiability condition is based upon, to avoid erroneous ambiguity detection — in Haskell, in the case of MPTCs — eliminates the need for the programmer to specify functional dependencies (FDs), although it has the drawback of removing from the programmer the possibility of using FDs in order to close the world. With the help of constraint projection, the world is closed automatically when (but only when) defined by the satisfiability-trigger condition. In our view, the use of constraint projection is an adequate tool for closing the world, releaving the programmer from the burden of specifying FDs. For example, the use of constraint projection would avoid the ambiguity that would be reported if FDs are removed from the class declaration below, since the type of g would be $SM\ m\ r \Rightarrow m\ a$.

```
class SM m r | m -> r, r -> m where
  { new:: a → m(r a); read:: r a → m a; write:: ...}

g x = do { r ← new x; read r }
```

8 Conclusion

We provide formal characterizations, in the context of constrained polymorphism, of open and closed worlds, constraint-set satisfiability and simplification, overloading resolution and type ambiguity. We define a type system for supporting both open and closed world approaches together and an algorithm for computing principal typings. Relative advantages between open and closed worlds are presented and briefly discussed, which suggest that both should be supported in a programming language. The theoretical framework and discussions presented provide a contribution to a better understanding of concepts and to further work and evolution of programming languages in this area.

References

1. Carlos Camarão, Lucília Figueiredo, and Cristiano Vasconcellos. Constraint-set Satisfiability for Overloading. In *Proc. of ACM PPDP'04*, pages 67–77, 2004.
2. K. Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, 1992.
3. Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. of POPL'82*, pages 207–212, 1982.
4. Fergus Henderson David Jeffery and Zoltan Zomogyi. Type Classes in Mercury. Technical Report 98/13, University of Melbourne, 1998.
5. Gregory J. Duck, Simon Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and Decidable Type Inference for Functional Dependencies. In *Proceedings of ESOP'2004 (European Symposium on Programming)*, 2004.

6. Dominic Duggan, Gordon Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Informatica*, 33(1):21–68, 1996.
7. Dominic Duggan and John Ophel. Open and closed scopes for constrained genericiy. *Theoretical Computer Science*, 275(1–2):215–258, 2002.
8. Dominic Duggan and John Ophel. Type checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):135–158, 2002.
9. Lucília Figueiredo and Carlos Camarão. Principal Typing and Mutual Recursion. In *Proc. WFLP'2001 (International Workshop on Functional and Logic Programming)*, pages 157–170, 2001.
10. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, 1996.
11. Fergus Henderson et al. The Mercury Project, 2003. http://www.cs.mu.oz.au/research/mercury.
12. Mark Jones. *Qualified Types*. Cambridge University Press, 1994.
13. Mark Jones. A system of constructor classes: overloading and higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–36, 1995.
14. Mark Jones. Simplifying and Improving Qualified Types. In *Proc. FPCA'95: ACM Conf. on Functional Prog. and Computer Architecture*, pages 160–169, 1995.
15. Mark Jones. Typing Haskell in Haskell. In *Proc. of Haskell Workshop'99, TR UU-CS-1999-28, Comp. Science Dept., Utrecht Univ.*, 1999. http://www.cse.ogi.edu/~mpj/thih.
16. Mark Jones. Type Classes with Functional Dependencies. In *Proc. of ESOP'2000*, 2000. Springer-Verlag LNCS 1782.
17. Mark Jones et al. Hugs98. http://www.haskell.org/hugs/, 1998.
18. Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994.
19. Simon P. Jones et al. GHC: The Glasgow Haskell Compiler. http://www.haskell.org/ghc.
20. Simon P. Jones et al. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
21. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
22. John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
23. Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *Journal of Functional Programming*, 1(1):1–100, 1993.
24. Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *Proc. of ACM Conference on Functional Programming and Computer Architecture*, pages 135–146, 1995.
25. Helmut Seild. Haskell Overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.
26. Geoffrey Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.
27. Martin Sulzmann et al. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 17(1), 2007.
28. Cristiano Vasconcelos, Lucília Figueiredo, and Carlos Camarão. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. *Journal of Universal Computer Science*, 9(8):873–890, 2003.
29. Dennis Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proc. ACM Symp. Functional Programming and Computer Architecture*, pages 15–28, 1991. LNCS 523.
30. Dennis M. Volpano. Haskell-style Overloading is NP-hard. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 88–94, 1994.
31. Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. of ACM POPL'89*, pages 60–76. ACM Press, 1989.